An Operational Semantics of First-class Synchronous Operations

John H. Reppy

TR 91-1232
August 1991

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Supported, in part, by the NSF and ONR under NSF grant CCR-85-14862, and by the NSF under NSF grant CCR-89-18233.
An Operational Semantics of
First-class Synchronous Operations

John H. Reppy *
Cornell University
jhr@cs.cornell.edu

August 15, 1991

Abstract

First-class synchronous operations are a new approach to synchronization and communication in concurrent languages. They have been informally described in [Rep88], [Rep91a]; this paper presents an operational semantics for an untyped language with first-class synchronous operations. This language includes a large fraction of the concurrency primitives of Concurrent ML (CML), a concurrent extension of SML, and is the first step toward formalizing the definition of CML.

1 Introduction

First-class synchronous operations are an abstraction mechanism for communication and synchronization in concurrent languages[Rep88, Rep91a]. The idea is to separate the description of synchronous communication from the act of synchronization (and communication), in much the same way that procedural abstraction separates the description of computation (i.e., λ-terms) from the actual computation (i.e., function application). The values used to represent synchronous operations are called events (we use this term instead of communications to stress the fact that synchronous operations other than channel I/O are covered by the mechanism).

First-class synchronous operations were originally developed as part of PML[Rep88], a concurrent ML-like language used in the Pegasus system. Later, the concurrency primitives of PML were implemented on top of SML/NJ, using first-class continuations[Rep89], CML, which evolved out of this implementation, extends and refines the concurrency mechanisms of PML[Rep91a]. Both PML and CML have been used for substantial implementations, which provide empirical evidence of the utility of first-class synchronous operations. Norman Ramsey independently developed a concurrency library on top of SML/NJ that

*Supported, in part, by the NSF and ONR under NSF grant CCR-85-14862, and by the NSF under NSF grant CCR-89-18233.
supports a variation of PML’s primitives\cite{Ram90}. Ramsey’s work illustrates an interesting fact: adding CSP-style message passing to SML will invariably result in something akin to first-class synchronous operations\footnote{This does not apply in the case of PML, since special syntax is being used to support the primitives.}. This is because selective communication must operate on multiple communications, which requires that there be type of values representing communications. This is not surprising, since the original motive for PML’s primitives was to provide an abstraction mechanism for channel I/O that cohabits with selective communication.

None of these previous efforts made any attempt to formalize the semantics of first-class synchronous operations; this paper is the first step towards remedying this situation. We present the operational semantics of a simple, untyped, call-by-value \(\lambda\)-calculus extended with first-class synchronous operations. This calculus, called \(\lambda_{cv}\), models most of the concurrency features of CML.

\subsection{Related work}

The semantics of Facile have been proposed as a model for PML in \cite{PGM90}, but no translation from PML to Facile is given. Independent work by Berry, Milner and Turner at the University of Edinburgh has resulted in an operational semantics for a small concurrent language, which includes the PML version of events\cite{BMT91}. The semantics presented in this paper has some strong similarities with that of \cite{BMT91}, but \(\lambda_{cv}\) is a richer language; in particular, the language of \cite{BMT91} does not include the guard and wrapAbort event value constructors or the poll operation of CML.

\subsection{Organization of this paper}

In the next section I present a brief overview of CML, including a description of its concurrency primitives. The following section presents \(\lambda_{cv}\), which models the core concurrency features of CML. In section 4, I discuss extending \(\lambda_{cv}\) to support additional features of CML. Finally, section 5 presents a polymorphic type system for \(\lambda_{cv}\), which uses SML’s imperative types. A more detailed discussion of these issues, including a proof of the soundness of the type system, can be found in the author’s forthcoming dissertation\cite{Rep91b}.

\section{A brief introduction to CML}

Before presenting the semantics of CML’s concurrency primitives, a brief review of the language is in order. A detailed overview of CML can be found in \cite{Rep91a}; the complete system is described in \cite{Rep90}, which includes a language tutorial.
CML is based on the sequential language SML\textsuperscript{MT90,MT91} and inherits the good features of SML: functions as first-class values, strong static typing, polymorphism, datatypes and pattern matching, lexical scoping, exception handling and a state-of-the-art module facility. The sequential performance of CML benefits from the quality of the SML/NJ compiler. In addition CML has the following properties:

- **CML** provides a high-level model of concurrency with dynamic creation of threads and typed channels, and rendezvous-style communication. This distributed-memory model fits well with the mostly applicative style of SML.

- **CML** is a higher-order concurrent language. Just as SML supports functions as first-class values, CML supports synchronous operations as first-class values. These values, called events, provide the tools for building new synchronization abstractions, which are tailored to the application.

- **CML** provides integrated I/O support. Potentially blocking I/O operations, such as reading from an input stream, are full-fledged synchronous operations. Low-level support is also provided, from which distributed communication abstractions can be constructed.

- **CML** provides automatic reclamation of threads and channels, once they become inaccessible. This permits a technique of speculative communication, which is not possible in other threads packages.

- **CML** uses pre-emptive scheduling. To guarantee interactive responsiveness, a single thread cannot be allowed to monopolize the processor. Pre-emption insures that a context switch will occur at regular intervals, which allows “off-the-shelf” code to be incorporated in a concurrent thread without destroying interactive responsiveness.

- **CML** is efficient. Thread creation, thread switching and message passing are very efficient (performance numbers are given in [Rep91a]). Experience with CML has shown that it is a viable language for implementing usable interactive systems.

- **CML** is portable. It is written in SML and runs on essentially every system supported by SML/NJ (currently four different architectures and many different operating systems).

To make this more concrete, Figure 1 gives the signature of the basic CML concurrency operations. These are also the concurrency operations of \( \lambda_{\text{cu}} \). CML programs consist of a collection of threads, which communicate via typed channels. Both threads and channels are created dynamically, using the functions \texttt{spawn} and \texttt{channel}, respectively. Rather than provide operations for communication, as is done in languages such as CSP\textsuperscript{Hoa78},
type 'a chan

val spawn : (unit -> unit) -> thread_id
val channel : unit -> 'a chan

val receive : 'a chan -> 'a event
val transmit : ('a chan * 'a) -> unit event

val choose : 'a event list -> 'a event
val guard : (unit -> 'a event) -> 'a event
val wrap : ('a event * ('a -> 'b)) -> 'b event
val wrapAbort : ('a event * (unit -> unit)) -> 'a event

val sync : 'a event -> 'a

Figure 1: Basic CML Concurrency Operations

occam and Amber, CML provides first-class values, called events, to represent synchronous operations. For example, the functions receive and transmit build event values to describe channel I/O operations. The function sync is used to actually apply these operations. There are also event combinators to build more complex synchronous operations:

choose. This constructs an event value that represents the non-deterministic choice of its arguments (note that this choice is made when sync is applied). A choice may involve multiple communications (both receive and transmit) on the same channel, but a thread cannot communicate with itself.

guard. This constructs an event out of an event valued function. When sync is applied, the function is called first, and the result is used for synchronization.

wrap. This wraps a function around an event value. If the event is chosen in a synchronization, then the function is applied to the result of the event.

wrapAbort. This associates an action to be taken if an event is not chosen in a synchronization. A new thread is spawned to execute the action.

The power of this approach is that it allows the user to implement new communication and synchronization abstractions. For example, we have found uses for widely varying abstractions, such as remote procedure call, multicast channels and buffered channels.

To illustrate these features, consider the CML program in Figure 2. This program first creates two channels, bound to ch1 and ch2; it then spawns two threads, which attempt to send messages on the channels. The parent thread synchronizes on the choice of receiving a message on either channel. Each arm of this choice has an associated wrapper and abort.
fun simple () = let
  val ch1 = channel() and ch2 = channel()
  val pr = CIO.print
  in
  pr "hi-0\n"
  spawn (fn () => (pr "hi-1\n"; send(ch1, 17); pr "bye-1\n"))
  spawn (fn () => (pr "hi-2\n"; send(ch2, 37); pr "bye-2\n"))
  sync (choose [
    guard (fn () => (pr "guard-0.1\n"
                     wrapAbort (wrap (receive ch1, fn _ => pr "bye-0.1\n"), fn () => pr "abort-0.1\n"))),
    guard (fn () => (pr "guard-0.2\n"
                     wrapAbort (wrap (receive ch2, fn _ => pr "bye-0.2\n"), fn () => pr "abort-0.2\n"))))
  ])
end

Figure 2: A simple CML program.

action; depending on which arm is selected, a different set of outputs is possible. The two graphs in Figure 3 detail the possible orderings of the output for the two choices. The dashed edges are induced by spawn, the dotted edges by message passing, and the solid edges by sequencing. These graphs illustrate the complementary relationship between the abort actions and wrapper functions. Note that the guards are always evaluated in left-to-right order.

Figure 4 gives the signature of some additional CML operations. These are discussed as extensions to $\lambda_{cv}$ (see Section 4). The function always takes a value $v$ builds a base event
val always : 'a -> 'a event
val threadWait : thread_id -> unit event

val accept : 'a chan -> 'a
val send : ('a * 'a chan) -> unit

datatype 'a option = NONE | SOME of 'a
val poll : 'a event -> 'a option

Figure 4: Other Concurrency Operations

value that is always immediately available for synchronization (yielding v). The function threadWait takes a thread identifier, such as one returned by spawn, and returns a base event value for synchronizing on the thread's termination. The operations accept and send are the channel I/O operations that the event values returned by receive and transmit represent. Lastly, the operation poll is a non-blocking form of sync; in a situation in which sync would block, it will return NONE instead of blocking (the option datatype is part of the SML/NJ pervasive environment).

3 The semantics of $\lambda_{cv}$

In this section I present the dynamic semantics of $\lambda_{cv}$, a concurrent extension of Plotkin's $\lambda_v$ calculus$^{[Plo75]}$. Although $\lambda_{cv}$ lacks many of the features of CML, it contains the core of the concurrency primitives (see Figure 1). In section 4, I describe how $\lambda_{cv}$ can be extended to include many of the missing features. The two features that are not described are the real-time sensitive operations (e.g., timeout) and I/O operations.

The dynamic semantics of Core CML are defined by two evaluation relations: a sequential evaluation relation $\rightarrow$, and a concurrent evaluation relation $\rightarrow\rightarrow$. Concurrent evaluation is an extension of sequential evaluation to finite sets of processes.

3.1 Syntax

We start with disjoint sets of variables, function constants, base constants and channel names:

\[
\begin{align*}
\mathcal{X} & \in \mathsf{VAR} & \text{variables} \\
\mathcal{B} & \in \mathsf{CONST} = \mathsf{FCONST} \cup \mathsf{BCONST} & \text{constants} \\
\mathsf{FCONST} & = \{+,-,\mathsf{fst}, \mathsf{snd}, \ldots\} & \text{function constants} \\
\mathsf{BCONST} & = \{(), \mathsf{true}, \mathsf{false}, 0, 1, \ldots\} & \text{base constants} \\
\kappa & \in \mathsf{CH} & \text{channel names}
\end{align*}
\]

To support events, it is required that $\mathsf{FCONST}$ include the following event value constructors:

\[
\text{choose, guard, never, receive, transmit, wrap, wrapAbort}
\]
There are three syntactic classes of terms in \( \lambda_{cv} \):

\[
\begin{align*}
\text{e} & \in \text{EXP} & \text{expressions} \\
\text{v} & \in \text{VAL} \subset \text{EXP} & \text{values} \\
\text{ev} & \in \text{EVENT} \subset \text{VAL} & \text{event values}
\end{align*}
\]

Values are the irreducible terms in the dynamic semantics. The terms of \( \lambda_{cv} \) are defined by the grammar in figure 5. Note that the syntactic class of the term \((v_1 . v_2)\) is either \text{EXP} or

\[
\text{e} ::= \text{v} \quad \text{value} \\
| \quad \text{e}_1 \; \text{e}_2 \quad \text{application} \\
| \quad (\text{e}_1 . \text{e}_2) \quad \text{pair} \\
| \quad \text{let} \; \text{x} = \text{e}_1 \; \text{in} \; \text{e}_2 \quad \text{let} \\
| \quad \text{chan} \; \text{x} \; \text{in} \; \text{e} \quad \text{channel creation} \\
| \quad \text{spawn} \; \text{e} \quad \text{process creation} \\
| \quad \text{sync} \; \text{e} \quad \text{synchronization}
\]

\[
\text{v} ::= \text{b} \quad \text{constant} \\
| \quad \text{x} \quad \text{variable} \\
| \quad (\text{v}_1 . \text{v}_2) \quad \text{pair value} \\
| \quad (\text{fn} \; \text{x} \mapsto \text{e}) \quad \text{\( \lambda \)-abstraction} \\
| \quad \kappa \quad \text{channels} \\
| \quad \text{ev} \quad \text{events} \\
| \quad (\text{G} \; \text{e}) \quad \text{guarded event function}
\]

\[
\text{ev} ::= \Lambda \quad \text{never} \\
| \quad \kappa! \text{v} \quad \text{channel output} \\
| \quad \kappa? \quad \text{channel input} \\
| \quad (\text{ev} \Rightarrow \text{v}) \quad \text{wrapper} \\
| \quad (\text{ev}_1 \oplus \text{ev}_2) \quad \text{choice} \\
| \quad (\text{ev} | \text{v}) \quad \text{abort wrapper}
\]

Figure 5: The \( \lambda_{cv} \) grammar

\( \text{VAL} \); this ambiguity is resolved in favor of \( \text{VAL} \). The value \( \Lambda \) is a base event value that is never matched (equivalent to \text{choose}[] in CML). The \( v \) subterms in the terms \((\text{ev} \Rightarrow \text{v})\) and \((\text{ev} | \text{v})\) are expected to be function values (the type system in Section 5 insures this). We write \( \text{VAL}^0 \) for the value terms without free variables; note, however, that closed values may contain free channel names. We write \( \text{FCV}(e) \) for the free channel names of a term \( e \). There is no term for sequencing, but we write \( "(e_1 ; e_2)" \) for \( \text{snd} (e_1 . e_2) \)," which, since the language is call-by-value, has the desired semantics. We adopt Barendregt's variable convention (p. 26 of [Bar84]) to avoid problems with capture.

Channel names and event values are not part of the concrete syntax of the language; rather, they appear as the intermediate results of evaluation. A \textit{program} is a closed term.
that does not contain any subterms of the form \((G e)\), or any subterms in \(\text{EVENT}\) or \(\text{CH}\) (in other words, programs do not have intermediate subterms).

### 3.2 Sequential evaluation

The meaning of the function constants is given by the partial function

\[
\delta : \text{FCONST} \times \text{VAL}^\circ \rightarrow \text{VAL}^\circ
\]

Since a closed value \(v \in \text{VAL}^\circ\) can have free channel names, we require that for all \(b \in \text{FCONST}\)

\[
\text{FCV}(\delta(b, v)) \subseteq \text{FCV}(v)
\]

For the standard built-in function constants, the meaning of \(\delta\) is the expected one. For example:

\[
\begin{align*}
\delta(\text{+}, (0.1)) &= 1 \\
\delta(\text{+}, (1.1)) &= 2 \\
\delta(\text{fst}, (v_1, v_2)) &= v_1 \\
\delta(\text{snd}, (v_1, v_2)) &= v_2
\end{align*}
\]

With the exception of \text{guard}, the event value constructors are mapped directly onto their corresponding terms in \text{EVENT}:

\[
\begin{align*}
\delta(\text{never}, ()) &= \Lambda \\
\delta(\text{transmit}, (\kappa, v)) &= \kappa!v \\
\delta(\text{receive}, \kappa) &= \kappa? \\
\delta(\text{wrap}, (ev, v)) &= (ev \Rightarrow v) \\
\delta(\text{choose}, (ev_1, ev_2)) &= (ev_1 \oplus ev_2) \\
\delta(\text{wrapAbort}, (ev, v)) &= (ev | v)
\end{align*}
\]

The \(\delta\)-rules for \text{guard} reflect its use as a delay operator; when another event constructor is applied to a guarded event value, then the guard operator \((G)\) is pulled out to delay the event construction. As is described below, the \text{sync} operator forces the evaluation of the guarded expression. In Algol 60 terminology, \((G e)\) is a \textit{thunk}.

\[
\begin{align*}
\delta(\text{guard}, v) &= (G (v () )) \\
\delta(\text{wrap}, ((G e), v)) &= (G (\text{wrap} (e, v))) \\
\delta(\text{choose}, ((G e_1), (G e_2))) &= (G (\text{choose} (e_1, e_2))) \\
\delta(\text{choose}, ((G e_1), ev_2)) &= (G (\text{choose} (e_1, ev_2))) \\
\delta(\text{choose}, (ev_1, (G e_2))) &= (G (\text{choose} (ev_1, e_2))) \\
\delta(\text{wrapAbort}, ((G e), v)) &= (G (\text{wrapAbort} (e, v)))
\end{align*}
\]

Following [WF91], the semantics is stated using \textit{evaluation contexts}. An evaluation context is a single hole context; the following grammar defines the evaluation contexts for \(\lambda_{ev}\):

\[
E ::= \[
| E e | v E | (E.e) | (v.E) |
\]

\[
\text{let } x = E \text{ in } e | \text{spawn } E | \text{sync } E
\]

8
These contexts force a call-by-value evaluation order. A term can be uniquely partitioned into an evaluation context and redex; we write $E[e]$ for this partition, where $e$ is the redex. We also use $E[e]$ for the term formed by inserting $e$, which might not be a redex, into the hole of $E$. The sequential evaluation relation ($\rightarrow$) is defined in terms of the contexts.

\[
\begin{align*}
E[b \; v] & \quad \rightarrow \quad E[\delta(b, v)] \\
E[(\text{fn } x \implies e) \; v] & \quad \rightarrow \quad E[e[x \mapsto v]] \\
E[\text{let } x = v \; \text{in } e] & \quad \rightarrow \quad E[e[x \mapsto v]] \\
E[\text{sync } G \; e] & \quad \rightarrow \quad E[\text{sync } e]
\end{align*}
\]

Note that the rule for sync forces the expression delayed by guard. We write $\rightarrow^*$ for the reflexive transitive closure of $\rightarrow$. As an example of sequential evaluation, consider the following, where $[\cdots]$ is used to mark the context/redex boundary.

\[
\begin{align*}
(f \; x \implies 1) \; ((f \; x \implies x \implies 10) \; (f \; y \implies y)) & \\
\rightarrow & \quad (f \; x \implies 1) \; ((f \; y \implies y) \; 10) \\
\rightarrow & \quad ((f \; x \implies 1) \; 10) \\
\rightarrow & \quad [1]
\end{align*}
\]

The derived syntax for sequencing leads to the following derived rule for evaluating a sequence of terms:

\[
E([v_1; \; v_2]) \rightarrow E[v_2]
\]

Note that both sides of the ";" are evaluated to values before this rule is applicable.

### 3.3 Event value matching

The core concept in the dynamic semantics of $\lambda_{ev}$ is the notion of event matching, which captures the semantics of rendezvous and communication. Informally, if two processes synchronize on matching events, then they can exchange values and continue evaluation. Before making this more formal, we need an auxiliary definition.

**Definition 3.1** The *abort action* of an event value, $ev$ is an expression, which, when evaluated, spawns the abort wrappers of $ev$. The map

\[
\text{AbortAct} : \text{EVENT} \rightarrow \text{EXP}
\]

is defined inductively on the structure of event values:

\[
\begin{align*}
\text{AbortAct}(\kappa?) & \quad = \quad () \\
\text{AbortAct}(\kappa!v) & \quad = \quad () \\
\text{AbortAct}(ev \rightarrow e) & \quad = \quad \text{AbortAct}(ev) \\
\text{AbortAct}(ev_1 \& ev_2) & \quad = \quad (\text{AbortAct}(ev_1); \; \text{AbortAct}(ev_2)) \\
\text{AbortAct}(ev \mid v) & \quad = \quad (\text{AbortAct}(ev); \; \text{spawn } v)
\end{align*}
\]
With this we can formally define the matching of event values:

**Definition 3.2 (Event matching)** We define a family of relations $\mathcal{R}_\kappa$, indexed by $\mathcal{CH}$, on $(\text{EVENT} \times \text{EXP})$. For $\kappa \in \mathcal{CH}$, we define

$$\text{ev}_1 \mathcal{R}_\kappa \text{ ev}_2 \text{ with } (e_1, e_2)$$

(pronounced as "$\text{ev}_1$ matches $\text{ev}_2$ on channel $\kappa$ with respective results $e_1$ and $e_2$") as the smallest relation satisfying the following rules:

1. $\kappa!v \mathcal{R}_\kappa ? v$ with $(\langle \rangle, v)$

2. If $\text{ev}_1 \mathcal{R}_\kappa \text{ ev}_2$ with $(e_1, e_2)$, then the following also hold:
   
   (2a) $\text{ev}_2 \mathcal{R}_\kappa \text{ ev}_1$ with $(e_2, e_1)$
   
   (2b) $\text{ev}_1 \mathcal{R}_\kappa (\text{ev}_2 \Rightarrow v)$ with $(e_1, v e_2)$
   
   (2c) $\text{ev}_1 \mathcal{R}_\kappa (\text{ev}_2 \oplus \text{ev}_3)$ with $(e_1, \text{AbortAct(} \text{ev}_3; e_2))$
   
   (2d) $\text{ev}_1 \mathcal{R}_\kappa (\text{ev}_3 \oplus \text{ev}_2)$ with $(e_1, \text{AbortAct(} \text{ev}_3; e_2))$
   
   (2e) $\text{ev}_1 \mathcal{R}_\kappa (\text{ev}_2 \mid v)$ with $(e_1, e_2)$

We write $\text{ev} \mathcal{R}_\kappa \text{ ev}'$ when we do not care about the results.

Informally, if two processes attempt to synchronize on matching event values, then the applications of $\text{sync}$ can be replaced with the respective results. This is made more precise in the next section, where the concurrent evaluation relation is defined.

It is worth noting that even if one of the wrappers of an event value is non-terminating, the necessary abort actions for that event will be executed (assuming fair evaluation). This property is important because a common CML idiom is to have tail-recursive calls in wrappers. For example, the following program merges two streams of input:

```plaintext
fun loop () = sync (choose [
    wrap (input_line instrm1, fn s => (print s; loop())),
    wrap (input_line instrm2, fn s => (print s; loop()))
])
```

Since the `input_line` operations are implemented using `wrapAbort` (see [Rep91a]) to abort the unwanted input request, it is important that the abort action be spawned, even though the wrapper never returns. Section 3.6 discusses this property in a more rigorous fashion.

### 3.4 Concurrent evaluation

The concurrent evaluation relation is an extension of the sequential evaluation relation $(\rightarrow)$ to finite sets of processes, with the addition of rules for process creation, termination and
communication. This is similar to the style of a “Chemical Abstract Machine” ([BB90]), except that there are no “cooling” or “heating” transitions ( configurations can be thought of as perpetually “hot” solutions). To avoid the complications of multisets, process identifiers are used to uniquely tag each process. The process identifier and term comprise a process; a finite set of processes is called a configuration:

\[ \pi \in \text{PROCID} \] 
\[ p = (\pi; e) \in \text{PROC} = (\text{PROCID} \times \text{EXP}) \] 
\[ \mathcal{P} \in \text{Fin(Proc)} \]

We often write a process as \( (\pi; E[e]) \), where the evaluation context serves the role of a program counter, marking the current state of evaluation.

**Definition 3.3** A configuration \( \mathcal{P} \) is well-formed if, for each \( (\pi; e) \in \mathcal{P} \), \( \text{FV}(e) = \emptyset \), and there is no \( e' \) such that \( (\pi; e') \in \mathcal{P} \).

It is occasionally useful to view well-formed configurations as finite maps from \( \text{PROCID} \) to \( \text{EXP} \), so we write \( \text{dom}(\mathcal{P}) \) for \( \{ \pi \mid (\pi; e) \in \mathcal{P} \} \) and \( \mathcal{P}(\pi) \) for \( e \) if \( (\pi; e) \in \mathcal{P} \). An initial configuration is a singleton process set \( \{(\pi; e)\} \), where \( e \) is a program. An initial configuration is trivially a well-formed one.

The concurrent evaluation relation is defined by rules of the form

\[ \mathcal{P} \rightarrow \mathcal{P}' \]

where \( \mathcal{P} \) and \( \mathcal{P}' \) are configurations. Most of these rules have enabling side conditions, so they are presented as inference rules. Each concurrent evaluation step affects one or two processes, called the selected processes. The left-associative operator +, is defined as \( \mathcal{P} + p \equiv \text{def} \mathcal{P} \cup \{p\} \), with \( \pi \notin \text{dom}(\mathcal{P}) \) for \( p = (\pi; e) \). This notation is used to distinguish the selected processes in the rules below. The first rule extends the sequential evaluation relation (\( \rightarrow \)) to configurations:

\[ \frac{e \rightarrow e'}{\mathcal{P} + (\pi; e) \rightarrow \mathcal{P} + (\pi; e')} \]

The creation of channels requires picking a new channel name and substituting for the variable bound to it:

\[ \kappa \notin \text{FCV}(\mathcal{P} + (\pi; E[\text{chan } x \text{ in } e])) \]
\[ \frac{\mathcal{P} + (\pi; E[\text{chan } x \text{ in } e]) \rightarrow \mathcal{P} + (\pi; E[e[x \rightarrow \kappa]])}{\mathcal{P} + (\pi; E[\text{spawn } v]) \rightarrow \mathcal{P} + (\pi; E[(\ell)] + (\pi'; v))} \]

Process creation requires picking a new process identifier:

\[ \pi' \notin \text{dom}(\mathcal{P}) + \pi \]
\[ \mathcal{P} + (\pi; E[\text{spawn } v]) \rightarrow \mathcal{P} + (\pi; E[(\ell)] + (\pi'; v)) \]

11
The most interesting rule describes communication and synchronization. If two processes are attempting synchronization on matching events, then they may rendezvous, exchange a message and continue evaluation:

\[ \text{ev} \overset{\kappa}{\approx} \text{ev}' \quad \text{with} \quad (e, e') \]

\[ \mathcal{P} + \langle \pi; E[\text{sync ev}] \rangle + \langle \pi'; E'[\text{sync ev}'] \rangle \quad \Rightarrow \quad \mathcal{P} + \langle \pi; E[e] \rangle + \langle \pi'; E'[e'] \rangle \]

We say that \( \kappa \) is used in this transition. Combined, these rules define concurrent evaluation. We write \( \Rightarrow^* \) for the reflexive transitive closure of \( \Rightarrow \).

### 3.5 An example

The following evaluation illustrates the concurrent evaluation relation:

\[
\{(\pi_0; [\text{chan } k \text{ in (spawn (fn } x \Rightarrow \text{sync (transmit (k .5))) ; sync (receive } k)])])
\]

\[
\Rightarrow \quad \{(\pi_0; [(\text{spawn (fn } x \Rightarrow \text{sync (transmit (} \kappa_0 .5))) ; \text{sync (receive } \kappa_0)])])
\]

\[
\Rightarrow \quad \{(\pi_0; (()); \text{sync ([receive } \kappa_0])]), (\pi_1; [(\text{fn } x \Rightarrow \text{sync (transmit (} \kappa_0 .5))) ()])
\]

\[
\Rightarrow \quad \{(\pi_0; (()); \text{sync ([receive } \kappa_0])]), (\pi_1; \text{sync ([transmit (} \kappa_0 .5)])])
\]

\[
\Rightarrow^* \quad \{(\pi_0; (()); [\text{sync (} \kappa_0!?)])]), (\pi_1; [\text{sync (} \kappa_0!5)])]
\]

\[
\Rightarrow \quad \{(\pi_0; ([(); .5]), (\pi_1; [(); 5]))
\]

\[
\Rightarrow \quad \{(\pi_0; [5]), (\pi_1; [(); 5])
\]

Note that this is only one of several possible evaluation sequences, although, in this example, all evaluation sequences produce the same result.

### 3.6 Some properties of evaluation

There are a few properties of concurrent evaluation that are worth noting.

**Lemma 3.1** If \( \mathcal{P} \) is a well-formed configuration and \( \mathcal{P} \Rightarrow \mathcal{P}' \), then \( \mathcal{P}' \) is well-formed. Likewise for \( \Rightarrow^* \).

**Proof.** The proof is by examination of the rules for \( \Rightarrow \). The rule for \text{spawn} is the only one that introduces process identifiers, and it only introduces new ones. Examination of the rules for \( \Rightarrow \), shows that it preserves closed terms. Thus \( \mathcal{P}' \) must be well-formed. This is extended to \( \Rightarrow^* \) by induction on the length of evaluation sequences.

Since initial configurations are well-formed, all of the interesting evaluations involve only well-formed configurations. Preservation of well-formedness is also shown in [BMT91] for their semantics. There are other properties of evaluation in [BMT91], which hold for \( \Rightarrow \).

**Lemma 3.2** If \( \mathcal{P} \) is well-formed and \( \mathcal{P} \Rightarrow \mathcal{P}' \), then the following hold:
• \( \text{dom}(P) \subseteq \text{dom}(P') \)
• \( \text{FCV}(P) \subseteq \text{FCV}(P') \)

• If \( \pi \notin \text{dom}(P') \), and \( \text{FV}(e) = \emptyset \), then \( P + \langle \pi; e \rangle \Rightarrow P' + \langle \pi; e \rangle \), with \( P' + \langle \pi; e \rangle \) being well-formed.

• If \( P = P'' + p \) and \( p \) is not selected in the transition to \( P' \), then there exists a \( P''' \), such that \( P' = P''' + p \).

As discussed on page 10, an important property of the semantics is the independence of abort operations from the wrappers of the chosen event. The following lemma characterizes the structure of the terms resulting from synchronization on events:

**Lemma 3.3** Let \( ev_1 \preceq ev_2 \) with \( (e_1, e_2) \). The terms \( e_1 \) and \( e_2 \) have the form of \( t \) specified by the following grammar:

\[
\begin{align*}
t & ::= v \\
& \mid (v \ t) \\
& \mid (a \ ; \ t) \\
\end{align*}
\]

\[
\begin{align*}
a & ::= () \\
& \mid (a_1 \ ; \ a_2) \\
& \mid (a \ ; \ \text{spawn} \ v)
\end{align*}
\]

In this grammar, the \( a \) subterms are contributed by \text{AbortAct}.

**Proof.** The lemma follows from definitions 3.1 and 3.2.

This structure, coupled with the call-by-value order of sequential evaluation, insures that the abort actions are spawned prior to application of any wrapper functions.

### 3.7 Fairness

The semantics presented above admits unfair evaluation sequences, and thus is not adequate as an specification of CML implementations. It is necessary to distinguish the acceptable evaluation sequences. Informally, we require that ready processes make progress and that communication on a single channel is fair (see [Kwi89] for a survey of fairness notions). To formalize these notions of fairness, we need a few definitions.

**Definition 3.4** A process \( p = \langle \pi; e \rangle \) is in one of three \textit{states}, depending on the form of \( e \):

• if \( e = [v] \), then \( p \) is a \textit{zombie},

• if \( e = E[\text{sync} \ ev] \), then \( p \) is \textit{blocked} on \( ev \),

• otherwise, \( p \) is \textit{ready}.
We define the set of ready processes in $\mathcal{P}$ by

$$\text{RDY}(\mathcal{P}) = \{ \pi \mid \langle \pi; \mathcal{P}(\pi) \rangle \text{ is ready} \}$$

A configuration $\mathcal{P}$ is terminal if every $p \in \mathcal{P}$ is a zombie or is blocked (i.e., deadlock).

**Definition 3.5** A process $p \in \mathcal{P}$ is enabled if $p \in \text{RDY}(\mathcal{P})$, or if it is blocked on $ev$ and there is a $p' \in \mathcal{P}$ blocked on $ev'$, with $ev \prec w ev'$.

We need a similar notion for channels.

**Definition 3.6** A channel $\kappa$ is enabled in a process configuration $\mathcal{P}$ if there are processes $\langle \pi; E[\text{sync } ev]\rangle, \langle \pi'; E'[\text{sync } ev']\rangle \in \mathcal{P}$, with $ev \prec w ev'$ and $\pi \neq \pi'$.

The acceptable evaluation sequences of a program are defined in terms of fairness restrictions.

**Definition 3.7** An evaluation sequence $T$ is acceptable if it is finite and ends in a terminal configuration, or if $T$ is infinite and satisfies the following fairness constraints:

1. Any process that is enabled infinitely often is selected infinitely often.
2. Any channel that is enabled infinitely often is used infinitely often.

In the taxonomy of [Kwi89], the first restriction is strong process fairness and the second is strong event fairness. Implementations are required to prohibit the possibility of unacceptable evaluation sequences, which, in practice, requires that implementations satisfy some stronger property on finite traces. As an example, consider the following property.

**Definition 3.8** A finite evaluation sequence $\mathcal{P}_0 \Rightarrow \mathcal{P}_1 \Rightarrow \cdots \Rightarrow \mathcal{P}_n$ is $k$-bounded fair (for $k$ a fixed positive integer), if every intermediate configuration $\mathcal{P}_i$, satisfies one of the following (where $m = i + k |\text{RDY}(\mathcal{P}_i)|$):

- $m > n$, or
- for every $\pi \in \text{RDY}(\mathcal{P}_i)$, $\pi$ is selected at least once in the evaluation subsequence $\mathcal{P}_i \Rightarrow \cdots \Rightarrow \mathcal{P}_m$.

An infinite evaluation sequence is $k$-bounded fair, if every finite prefix of it is $k$-bounded fair.
An infinite evaluation sequence that is $k$-bounded fair obviously satisfies restriction (1) (but not necessarily (2)). This restriction is realizable using fairly standard implementation techniques. For example, an implementation that uses fair pre-emptive scheduling\(^2\) and FIFO queues for the process ready queue and for channel waiting queues will produce only $k$-bounded fair sequences, where $k$ is determined by the length of the time-slice and speed of the processor.

4 Extending $\lambda_{cv}$ to full CML

The language $\lambda_{cv}$ lacks a number of features found in CML; in this section we examine how the semantics of $\lambda_{cv}$ can be extended to model some of these features.

4.1 Syntactic sugar

There are a number of missing CML features that can be viewed as syntactic sugar for $\lambda_{cv}$ terms. CML uses the function channel to allocate new channels and provides the more traditional synchronous operations send and accept. These functions can be used by embedding a $\lambda_{cv}$ term $e$ in the following context:

\[
\begin{align*}
\text{let channel} &= \text{fn } x \Rightarrow \text{`chan } k \text{ in } k \text{ in} \\
\text{let send} &= \text{fn } x \Rightarrow \text{sync (transmit } x \text{) in} \\
\text{let accept} &= \text{fn } x \Rightarrow \text{sync (receive } x \text{) in} \\
\end{align*}
\]

\[ [e] \]

The choose and select functions of CML work on lists of events (instead of just pairs). Although $\lambda_{cv}$ does not have SML's datatypes, event lists can be implemented using the following translation:

\[
\begin{align*}

[\text{nil}] &= \text{never()} \\
[\text{ev}::r] &= \text{choose (ev,}[r]\text{)}
\end{align*}
\]

4.2 Recursion

Dynamic process and channel creation is powerful enough to implement the call-by-value $Y_e$ combinator, which has the following evaluation rule:

\[ E[Y_e v] \rightarrow E[v (\text{fn } x \Rightarrow (Y_e v) \ x)] \]

The following CML code implements $Y_e$ (adopted from [GMP89]):

\(^2\text{By fair, I mean that a thread is guaranteed some progress before being pre-empted.}\)
val \( Y_s \) = fn f \rightarrow let
val a = channel()
val g = fn v \rightarrow let val h' = accept a
in
spawn (fn () \rightarrow send(a, h'))
f h' v
end
in
spawn (fn () \rightarrow send(a, g));
let val h = accept a
in
spawn (fn () \rightarrow send(a, h));
f h
end
end

This code is somewhat mysterious, but what it actually does is fairly simple. The channel a is used to cache the function g for the next iteration of f; each time g (renamed h) is read from a, a new thread is spawned to send another copy. For CML, which is statically typed (see section 5), this definition implements recursion at all imperative types. As an alternative, we could add the \( Y_s \) combinator as a built-in term constructor (as is done in [WF91]), which would provide recursion at all types.

### 4.3 References

It is well known that processes and channels can be used to mimic updatable references. The following CML code illustrates this:

```plaintext
datatype 'a ref = REF of ('a chan * 'a chan)
fun mkRef initX = let
  val inCh = channel() and outCh = channel()
  fun cell x = sync (choose [
    wrap (transmit (outCh, x), fn () \rightarrow cell x),
    wrap (receive inCh, fn newX \rightarrow cell newX)
  ])
in
  spawn (fn () \rightarrow cell initX);
  REF(inCh, outCh)
end

fun assign (REF(inCh, _), x) = send (inCh, x)

fun deref (REF(_, outCh)) = accept outCh
```

One can define a formal translation from programs with references to programs using this scheme. This is done in [BMT91], and the translation is shown to be semantics preserving.

The implementation of \( Y_s \) described in section 4.2 is similar to the imperative \( Y \)-combinator (\( Y_i \)) defined by Felleisen[Fei87]. This suggests the following implementation of references, which uses channels to represent references directly and does not require explicit recursion:
fun mkRef initX = let val ch = channel() in
  spawn (fn () => send (ch, initX));
  ch
end

fun assign (ch, x) = (accept ch; spawn (fn () => send (ch, x)))

fun deref ch = let val x = accept ch in
  spawn (fn () => send (ch, x));
  x
end

4.4 Exceptions

CML inherits SML's exception mechanism and provides an event combinator for handling exceptions that are raised during evaluation of an events wrappers (wrapHandler). Wright and Felleisen provide a semantics of SML's exception mechanism in [WF91], but applying this technique to λce will require some care. The problem is that the soundness of their semantics relies on limiting the scope of exception identifiers to within the scope of their binding site (the rewrite rules allow the binding sites to migrate up to the top of the term thus expanding the scope of the binding). Since processes can include exceptions in messages, a different approach is needed. The best approach seems to be to bind exception identifiers in an implicit global environment (as is done with channel names). New syntax for expressions is required:

\[
\begin{align*}
  e &::= e \text{ handle } v \quad \text{exception handler} \\
    &\mid \text{raise } x\ e \quad \text{raise exception} \\
    &\mid \ldots \\
  \hat{e}v &::= (ev H v) \quad \text{handler wrapper} \\
    &\mid \ldots
\end{align*}
\]

The event constructor wrapHandler has the following δ-rule:

\[
\delta(\text{wrapHandler}, (ev.v)) = (ev H v)
\]

A new clause is needed in Definition 3.2:

(2f) \(ev_1 \triangleright (ev_2 H v)\) with \((e_1, e_2 \text{ handle } v)\)

There are also new evaluation contexts and rules; see [WF91] for details.

4.5 Process join

CML provides the event constructor threadWait that allows a thread to synchronize on the termination of another thread. One approach to describing this is to add a set of
distinguished channel names, \( \{ \kappa_\pi \mid \pi \in \text{PROCID} \} \), to represent processes. The rule for process creation wraps the body of a process \( \pi \) with code to repeatedly send \( () \) on the channel \( \kappa_\pi \):

\[
\frac{\pi' \notin \text{dom}(P)}{P + (\pi; E[\text{spawn } v]) \Longrightarrow P + (\pi; E[\kappa_\pi]) + (\pi'; \text{Fork}(\pi', v))}
\]

where

\[
\text{Fork}(\pi, v) = (v (); Y_x (fn f \Rightarrow (\text{send } (\kappa_\pi . ()) ; f () ) ()))
\]

Then (threadWait \( \kappa_\pi \)) is implemented by (accept \( \kappa_\pi \)). While this is a reasonable implementation technique, it has the problem that it becomes much harder to distinguish the zombie processes in the semantics.

A better approach is to directly support threadWait and, while we are at it, also support the base event constructor always. To do so we add PROCID to the domain of values and add two new event value terms:

\[
v ::= \pi | \cdots
\]

\[
ev ::= (W \pi) | A | \cdots
\]

The implementation of the always function is defined by the following \( \delta \)-rule:

\[
\delta(\text{always}, v) = (A \Rightarrow (fn x \Rightarrow v))
\]

The rule for spawn is slightly changed to return the identifier of the new process:

\[
\frac{\pi' \notin \text{dom}(P)}{P + (\pi; E[\text{spawn } v]) \Longrightarrow P + (\pi; E[\pi]) + (\pi'; v ()})
\]

We then need to introduce a notion of matching an event by a set of processes:

**Definition 4.1** We define a ternary relation

\( ev \subseteq P \) with \( e \)

(pronounced as "\( ev \) is matched by \( P \) with result \( e \)"") as the smallest relation satisfying the following rules:

1. \( (W \pi) \subseteq P + (\pi; [v]) \) with \( () \)
2. \( A \subseteq P \) with \( () \)
3. If \( ev \subseteq P \) with \( e \), then the following also hold:
   3a. \( (ev \Rightarrow v) \subseteq P \) with \( (v e) \)
   3b. \( (ev \oplus ev') \subseteq P \) with \( (\text{AbortAct}(ev'); e) \)
(3c) \( (ev' \oplus ev) \subseteq \mathcal{P} \) with \( \text{AbortAct}(ev') \); \( e \)

(3d) \( (ev \mid v) \subseteq \mathcal{P} \) with \( e \)

And there is an additional rule for \( \text{sync} \):

\[
\text{ev} \subseteq \mathcal{P} \text{ with } e \\
\overline{\mathcal{P} + (\pi; \text{sync ev}) \rightarrow \mathcal{P} + (\pi; e)}
\]

4.6 Polling

Both PML and CML support polling of events\(^3\). The poll operation is now a non-blocking form of \( \text{sync} \) with the type:

\[
\text{val poll} : \text{'a event} \rightarrow \text{'a option}
\]

Applying \( \text{poll} \) to an event value returns \( \text{NONE} \) if \( \text{sync} \) would have blocked, and \( \text{SOME} \) wrapped around the synchronization result otherwise.

It is fairly straightforward to add \( \text{poll} \) to \( \lambda_{ev} \). Since we do not have the \text{option} datatype in \( \lambda_{ev} \), we have the \( \text{poll} \) operation take two arguments: an event value and a pair of functions; the first is evaluated if the event is matched and the second is evaluated otherwise, so \( \text{poll} \) has the following type:

\[
\text{val poll} : (\text{'a event} \times (((\text{'a} \rightarrow \text{'b}) \times (\text{unit} \rightarrow \text{'b}))) \rightarrow \text{'b}
\]

We extend the syntax of expression terms and the definition of evaluation contexts by

\[
e := \text{poll e} | \cdots \\
E := \text{poll E} | \cdots
\]

We also need to formalize the notion that an event is matched by some other offered event in a configuration. The following two definitions do this.

**Definition 4.2** We say that \( ev \) is offered by \( \pi \) in a configuration \( \mathcal{P} \), if \( \mathcal{P}(\pi) \) is of the form \( E[\text{sync ev}] \) or \( E[\text{poll}(ev . v)] \). We define the set of offered events of \( \mathcal{P} \) by

\[
\text{Offered}(\mathcal{P}) = \{ (\pi, ev) \mid \pi \text{ offers } ev \text{ in } \mathcal{P} \}
\]

\(^3\)In PML polling was supported via an awkward priority scheme on base event values. While in [Rep91a] polling was presented as a event-value constructor:

\[
\text{val poll} : \text{'a event} \rightarrow \text{'a option event}
\]

In practice, however, the full generality of this mechanism was not used, so CML now uses the simpler mechanism described in this paper.
Definition 4.3 We say that an event $ev$ is matched in a set of processes $\mathcal{P}$, written $ev \mathcal{\subseteq} \mathcal{P}$, if there exists $(\pi', ev') \in \text{Offered}(\mathcal{P})$ such that $ev \mathcal{\subseteq} ev'$.

And, we need three additional concurrent evaluation rules. The first two handle the transition in which the event is matched by some other process, the third handles the transition for when sync would have blocked:

\[
\frac{ev \mathcal{\subseteq} ev' \text{ with } (e, e')}{\mathcal{P} + (\pi; E[poll(ev.(v_1.v_2))]) \rightarrow \mathcal{P} + (\pi'; E'[\text{sync } ev'])} \rightarrow \mathcal{P} + (\pi; E[v_1 e]) + (\pi'; E'[e'])
\]

\[
\frac{ev \mathcal{\subseteq} ev' \text{ with } (e, e')} {\mathcal{P} + (\pi; E[poll(ev.(v_1.v_2))]) \rightarrow \mathcal{P} + (\pi'; E'[poll(ev'.(v'_1.v'_2))])} \rightarrow \mathcal{P} + (\pi; E[v_1 e]) + (\pi'; E'[v'_1 e'])
\]

\[
\frac{ev \not\mathcal{\subseteq} \mathcal{P}} {\mathcal{P} + (\pi; E[poll(ev.(v_1.v_2))]) \rightarrow \mathcal{P} + (\pi; E[v_2 ()])}
\]

To make these rules sensible, we need an additional fairness constraint.

If $p = (\pi; E[poll(ev.v)])$, then a transition $\mathcal{P} + p \rightarrow \mathcal{P}' + p$, is acceptable if:

- $ev \mathcal{\subseteq} \mathcal{P}$ and $ev \mathcal{\subseteq} \mathcal{P}'$, or
- $ev \not\mathcal{\subseteq} \mathcal{P}$ and $ev \not\mathcal{\subseteq} \mathcal{P}'$.

This constraint captures the notion that poll is non-blocking by forcing the polling operation to complete before the state of the polled event can change.

5 Typing $\lambda_{cv}$

The fact that references can be coded up using channels makes apparent the fact that polymorphic channels incur the same typing problems as polymorphic references (see [Tof90] for a good description of these problems). It is possible, however, to soundly type $\lambda_{cv}$ programs using the imperative type scheme of SML$^{\text{IMPTYVAR}}$ [MTH90, Tof90]. In this section, I present a type system for $\lambda_{cv}$ programs (but not intermediate terms). I claim, without proof, that this system is sound with respect to the dynamic semantics given in section 3. The sound typing of $\lambda_{cv}$ is discussed in greater detail (including a proof of the soundness of the type system) in [Rep91b].

The presentation uses standard notation (e.g., see [Tof90] or [WF91]). We start with type constants (int, bool, etc.); we use $\iota$ to range over type constants. Type variables are partitioned into two sets: imperative type variables (IMPTYVAR) and applicative type
variables \(\text{APPTYVAR}\); we use \(\alpha\) to range over both kinds of type variables. The set of types, \(\tau \in \text{TY}\), and type schemes, \(\sigma \in \text{TYSHEME}\), are defined by

\[
\tau ::= \iota \quad \text{type constants} \\
| \alpha \quad \text{type variables} \\
| (\tau_1 \to \tau_2) \quad \text{function types} \\
| (\tau_1 \times \tau_2) \quad \text{pair types} \\
| \tau \text{ chan} \quad \text{channel types} \\
| \tau \text{ event} \quad \text{event types}
\]

\[
\sigma ::= \tau \\
| \forall \alpha.\sigma
\]

We write \(\text{FTV}(\tau)\) for the free type variables of \(\tau\). A type is imperative if its free type variables are all imperative; we use \(\theta\) to range over imperative types. We write \(\sigma \succ \tau\) if \(\tau\) is an instance of \(\sigma\). A type environment \(\text{TE}\) is a finite function from variables to type schemes. We write \(\text{FTV}(\text{TE})\) for the free type variables of a type environment. The closure of a type \(\tau\) with respect to the type environment \(\text{TE}\) is defined as

\[
\text{CLOS}_{\text{TE}}(\tau) = \forall \alpha_1 \ldots \alpha_n.\tau
\]

where \(\{\alpha_1, \ldots, \alpha_n\} = \text{FTV}(\tau) \setminus \text{FTV}(\text{TE})\), and the applicative closure of \(\tau\) by

\[
\text{APP}\text{CLOS}_{\text{TE}}(\tau) = \forall \alpha_1 \ldots \alpha_n.\tau
\]

where \(\{\alpha_1, \ldots, \alpha_n\} = (\text{FTV}(\tau) \setminus \text{FTV}(\text{TE})) \cap \text{APPTYVAR}\). To assign types to constants we assume the existence of the function

\[
\text{TypeOf} : \text{CONST} \to \text{TYSHEME}
\]

In addition to mapping 1 to \text{int}, etc., the TypeOf function assigns types to the various concurrency related constants; Figure 6 gives these types. Note that we treat the syntactic

\[
\begin{align*}
\text{never} &: \forall \alpha. (\text{unit} \to \alpha \text{ event}) \\
\text{spawn} &: \forall \alpha. ((\text{unit} \to \text{unit}) \to \text{unit}) \\
\text{sync} &: \forall \alpha. (\alpha \text{ event} \to \alpha) \\
\text{transmit} &: \forall \alpha. ((\alpha \text{ chan} \times \alpha) \to \text{unit} \text{ event}) \\
\text{receive} &: \forall \alpha. (\alpha \text{ chan} \to \alpha \text{ event}) \\
\text{guard} &: \forall \alpha. ((\text{unit} \to \alpha \text{ event}) \to \alpha \text{ event}) \\
\text{wrap} &: \forall \alpha\beta. ((\alpha \text{ event} \times (\alpha \to \beta)) \to \beta \text{ event}) \\
\text{wrapAbort} &: \forall \alpha. ((\alpha \text{ event} \times (\text{unit} \to \text{unit})) \to \alpha \text{ event}) \\
\text{choose} &: \forall \alpha. ((\alpha \text{ event} \times \alpha \text{ event}) \to \alpha \text{ event})
\end{align*}
\]

Figure 6: Typing of concurrency constants
forms of sync and spawn as applications of constants in the typing system. The typing judgements are sentences of the form

\[ \text{TE} \vdash e : \tau \]

which are read as "e has type \( \tau \) under the type assumptions of TE." The typing rules are given in Figure 7. A program \( e \) is well-typed if we can deduce \( \{ \} \vdash e : \tau \), for some \( \tau \), from

\[
\frac{\text{TypeOf}(b) \supset \tau}{\text{TE} \vdash b : \tau} \quad (\tau\text{-const})
\]

\[
\frac{x \in \text{dom}(\text{TE})}{\text{TE} \vdash x : \tau} \quad (\tau\text{-var})
\]

\[
\frac{\text{TE} \vdash e_1 : (\tau' \rightarrow \tau) \quad \text{TE} \vdash e_2 : \tau'}{\text{TE} \vdash e_1 \ e_2 : \tau} \quad (\tau\text{-app})
\]

\[
\frac{\text{TE} \vdash \{ x \mapsto \tau \} \vdash e : \tau'}{\text{TE} \vdash (\text{fn } x \Rightarrow e) : (\tau \rightarrow \tau')} \quad (\tau\text{-abs})
\]

\[
\frac{\text{TE} \vdash e_1 : \tau_1 \quad \text{TE} \vdash e_2 : \tau_2}{\text{TE} \vdash (e_1 , e_2) : (\tau_1 \times \tau_2)} \quad (\tau\text{-pair})
\]

\[
\frac{\text{TE} \vdash v : \tau' \quad \text{TE} \vdash \{ x \mapsto \text{CLOS}_{\text{TE}}(\tau') \} \vdash e : \tau}{\text{TE} \vdash \text{let } x = v \text{ in } e : \tau} \quad (\tau\text{-let-v})
\]

\[
\frac{\text{TE} \vdash e_1 : \tau' \quad \text{TE} \vdash \{ x \mapsto \text{APP}_{\text{CLOS}_{\text{TE}}(\tau')} \} \vdash e_2 : \tau}{\text{TE} \vdash \text{let } x = e_1 \text{ in } e_2 : \tau} \quad (\tau\text{-let-e})
\]

\[
\frac{\text{TE} \vdash \{ x \mapsto \emptyset \text{ chan} \} \vdash e : \tau}{\text{TE} \vdash \text{chan } x \text{ in } e : \tau} \quad (\tau\text{-chan})
\]

Figure 7: Typing rules for \( \lambda_{\text{cv}} \) programs.

The typing rules are fairly standard, but it is worth commenting on a few of them. There are two rules for let expressions: (\( \tau\text{-let-v} \)), which applies in the case that the variable is being bound to a non-expansive expression; and (\( \tau\text{-let-e} \)), which applies when the variable is being bound to an expression that is expansive (i.e., one that might introduce new channels). Expansiveness is a syntactic property; in our notation, the non-expansive expressions are exactly the value terms. The other interesting rule is (\( \tau\text{-chan} \)), which restricts the type of variables bound to the created channel to be imperative. These rules combine to prevent type loopholes, while allowing some polymorphic treatment of channels. For a more detailed discussion of imperative types see [Tof90].


6 Future work

This paper describes the dynamic semantics of a large fraction of the concurrency features of CML. There are two obvious directions to continue work: first, to show the soundness of the typing system given in section 5; and, second, to expand the semantics to cover the whole of CML. Work is almost complete on the first of these tasks, and will be included in [Rep91b]. Another useful direction would be to build a “theory” of CML programs to allow reasoning about their correctness.

Acknowledgements

Bill Aitken, Bard Bloom, Dave MacQueen and Chet Murthy have all provided helpful comments on earlier drafts of this paper.

References


